# Lambda Expressions and Closures: Wording for Monomorphic Lambdas

Document no: N2413=07-0273

Jaakko Järvi* Texas A&M University

John Freeman Texas A&M University

Lawrence Crowl Google Inc.

2007-09-09

## 1 Introduction

This document proposes *lambda expressions* to be adopted to C++. The proposal revises N2329 [JFC07]. N2329 was a major revision of the document N1968 [WJG+06], and draw also from the document N1958 [Sam06] by Samko. The differences to N2329 are:

- This document provides wording for monomorphic (non-generic) lambda functions, which we propose for the committee to move forward with for C++0X. The discussion on polymorphic (generic) lambda functions is not included in this document. Following the committee's recommendation from the Toronto meeting (July 2007), we separate the specification of polymorphic lambda functions into a separate (later) document. Polymorphic lambda functions can be made upwards compatible with the non-generic lambda functions proposed here.
- There are small changes in the proposed syntax.
- Declaring new variables to be stored in a closure "in place" in the lambda expression is no longer supported, instead, only local variables in the scope of the lambda expression can be declared as closure members. The previously proposed more general syntax is upwards compatible.
- The translation to function objects that defines the semantics of lambda functions now translates to local classes, anticipating a change in the standard that will allow local classes to be used as template arguments.
- Discussions on alternative syntaxes and design choices are not included; for those, we refer the reader to document N2329 [JFC07].
- Discussion on "concept map dropping" problem is not included as it is not relevant for monomorphic lambda functions.

We use the following terminology in this document:

- *Lambda expression* or *lambda function*: an expression that specifies an anonymous function object
- *Closure*: An anonymous function object that is created automatically by the compiler as the result of evaluating a lambda expression. Closures consists of the code of the body of the lambda function and the *environment* in which the lambda function is defined. In practice this means that variables referred to in the body of the lambda function are stored as member variables of the anonymous function object, or that a pointer to the frame where the lambda function was created is stored in the function object.

*jarvi@cs.tamu.edu

The proposal relies on several future additions to C++, some of which are already in the working draft of the standard, others likely candidates. These include the **decltype** [JSR06b] operator, new function declaration syntax [JSR06a, Section 3], and changes to linkage of local classes [Wil07].

## 2 In a nutshell

The use of function objects as higher-order functions is commonplace in calls to standard algorithms. In the following example, we find the first employee within a given salary range:

```
class between {
  double low, high;
public:
  between(double l, double u) : low(l), high(u) { }
  bool operator()(const employee& e) {
    return e.salary() >= low && e.salary() < high;
  }
}
...
double min_salary;
...
std::find_if(employees.begin(), employees.end(),
             between(min_salary, 1.1 * min_salary));
```

The constructor call between(min_salary, 1.1 ∗ min_salary) creates a function object, which is comparable to a *closure*, a term commonly used in the context of functional programming languages. A closure stores the *environment*, that is the values of the local variables, in which a function is defined. Here, the environment stored in the between function object are the values low and high, which are computed from the value of the local variable min_salary. Thus, we store the information necessary to evaluate the body of the **operator**()() in the closure.

The syntactic requirement of defining a class with its member variables, function call operator, and constructor, and then constructing an object of that type is very verbose, and not well-suited for creating function objects "on the fly" to be used only once. The essence of this proposal is a concise syntax for defining such function objects—indeed, we define the semantics of lambda expressions via translation to function objects. With the proposed features, the above example becomes:

```
double min_salary = ...
...
double u_limit = 1.1 * min_salary;
std::find_if(employees.begin(), employees.end(),
             <&>(const employee& e) (e.salary() >= min_salary && e.salary() < u_limit));
```

## 3 About non-generic and generic lambda functions

The lambda expression:

```
<&>(const employee& e) (e.salary() >= min_salary && e.salary() < u_limit)
```

is *monomorphic*, because the types of its parameters are explicitly specified. Here, the type of the only parameter e has type **const** employee&. A *polymorphic* version of the same expression would be written as:

```
<&>(e) (e.salary() >= min_salary && e.salary() < u_limit)
```

The latter form requires that the parameter types are deduced (from the use of the lambda), and have a substantially higher implementation cost. As said, this proposal does not discuss generic lambda functions.

# 4 Proposal

In the following, we introduce the proposed feature informally using examples of increasing complexity.

## 4.1 Lambda functions with no external references

We first discuss lambda functions that have no references to variables defined outside of its parameter list. We demonstrate with a binary lambda function that invokes **operator**+ on its arguments. The most concise way to define that lambda function is as follows:

```
<>(int x, int y) ( x + y )
```

This lambda function can only be called with arguments that are of type **int**, or convertible to type **int**. In this examples, the body of the lambda function is a single expression, enclosed in parentheses. The return type does not have to be specified; it is deduced to be the type of the expression comprising the body. Return type deduction is defined with the help of the **decltype** operator. Above, the return type is defined as **decltype**(x + y).

Explicitly stating the return type is allowed as well, for which the syntax is as follows:

```
<>(int x, int y) -> int ( x + y )
```

It is possible to define lambda functions where the body is a block of statements, rather than a single expression. In that case, the return type must be specified explicitly:

```
<>(int x, int y) -> int { int z; z = x + y; return z; }
```

There are several reasons for requiring the return type to be specified explicitly in the case where the body consist of a block statement. First, the body of a lambda function could contain more than one return statement, and the types of the expressions in those return statements could differ. Such definitions would likely have to be flagged out as ambiguous, or rules similar to those of the conditional operator could be applied, which is potentially complicated. Second, implementing return type deduction from a statement block may be non-trivial. The return type is no longer dependent on a single expression, but rather requires analyzing a series of statements, possibly including variable declarations etc. Third, while performing overload resolution, lambda function's return type expression is instantiated; to avoid hard errors during overload resolution, certain errors in the return type expression should fall under the SFINAE rules. Whether an arbitrary block type checks or not as a SFINAE condition is not feasible, nor desirable.

The semantics of lambda functions are defined by translation to function objects. For example, the last lambda function above is defined to be behaviorally equivalent with the function object below. The proposed translation is somewhat more involved; we describe the full translation in Section 5.

```
class F {
public:
  F() {}
  auto operator()(int x, int y) const -> int {
    int z; z = x + y; return z;
  }
};
F() // create the closure
```

We summarize the rules this far:

- Each parameter slot in the parameter list of a lambda function must of an "ordinary" function parameter declaration with a non-abstract declarator. A type without a parameter name is not allowed to preserve upwards compatibility to polymorphic lambda functions, where a single identifier is interpreted as the name of the parameter, not a type.
- The body of the lambda function can either be a single expression enclosed in parenthesis or a block.

- If the body of the lambda function is a block, the return type of the lambda function must be explicitly specified.
- If the body of the lambda function is a single expression, the return type of the lambda function may be explicitly specified. If it is not specified, it is defined as **decltype**(e), where e is the body of the lambda expression.

## 4.2 External references in lambda function bodies

References to local variables declared outside of the lambda function bodies have been the topic of much debate. Any local variable referenced in a lambda function body must somehow be stored in the resulting closure. The earlier proposal [WJG+06] called for storing such local variables by copy and required an explicit declaration to instruct that a variable should be stored by reference instead. Another proposal by Samko [Sam06] suggested the opposite. However, neither alternative gained wide support as both approaches have notable safety problems. By-reference can lead to dangling references, by-copy to unintentional slicing of objects, expensive copying, invalidating iterators, and other surprises.

We propose that neither approach is the default and require the programmer to explicitly declare whether by-reference or by-copy is desired. There are two mechanisms for this. Either specifying "by-reference" for the entire lambda, or by specifying one of the two modes for each variable stored into the closure We discuss the latter mechanism first.

All local variables referred to in the body of the lambda function (but define outside of it) must be declared alongside the parameters of the lambda function. These declared local variables are what are stored in the closure. The following example defines a lambda function, where the closure stores a reference to the local variable sum, and stores a copy of a local variable factor:

```
double array[] = { 1.0, 2.1, 3.3, 4.4 };
double sum = 0; int factor = 2;
for_each(array, array + 4, <>(double d ; &sum, factor) ( sum += factor * d ));
```

We refer to the part of the function signature that declares the member variables of the closure as the lambda function's *local variable clause*. It is a comma separate list of unqualified identifiers, optionally preceded with &, that name a variable with a non-static storage duration. The closure stores references to objects whose identifiers are declared with & and copies of objects whose identifiers are declared without &.

To clarify how local variables are handled, the above lambda function is behaviorally equivalent to the following function object:

```
class F {
  double& sum;
  mutable int factor;
public:
  F(int& sum, int factor) : sum(sum), factor(factor) {}
  auto operator()(double d) const -> decltype(sum += factor * d) {
    return sum += factor * d;
  }
};
F(sum, factor); // create the closure
```

Here we rely on a recent change in the standard that brings the member variable names sum and factor in scope in the return type of **operator**().

The variable **this** requires special handling. If a lambda function is defined in a member function, the body of the lambda function may contain references to **this**. To allow these references, **this** must be explicitly declared in the local variable clause. In the translation, references to **this** in the body of the lambda function are retained to refer to the object in whose member function the lambda was defined, not the just generated closure object. The variable **this** in the local variable clause causes some unique member variable, call it __this, to be created to store the value of **this**. All references to **this**, including implicit member access

operator calls that do not mention **this** explicitly, are then translated to references to __this.The type of this_ will be appropriately qualified depending on the cv-qualifiers of the member function the lambda is defined in. Also, the newly generated closure should be a **friend** of the enclosing class to allow access to its private members. The following example demonstrates the handling of **this**, and the necessity of making the lambda function a **friend** of the enclosing class:

```
class A {
  vector<int> v;
  ...
public:
  void zero_all(const vector<int>& indices) {
    for_each(indices.begin(), indices.end(), <>(int i : this) ( this-> v[i] = 0 ));
  }
};
```

Requiring explicit declaration of the variables that are to be stored into closure has the benefit, over the previous proposals, that the programmer is *forced* to express his or her intention on what storage mechanism should be used for each local variable. The disadvantage is verbosity.

We suspect a common use of lambda functions is as function objects to standard algorithms and other similar functions, where the lifetime of the lambda function does not extend beyond the lifetime of its definition context. In such cases, it is safe to store the environment into a closure by reference. For this purpose, we suggest syntax that allows the "by-reference" declaration to be made for the entire lambda at once, and not requiring explicitly listing variable names in a local variable clause. Rewriting our previous example some, the two calls to for_each below are equivalent:

```
double array[] = { 1.0, 2.1, 3.3, 4.4 };
double sum = 0; int factor = 2;
for_each(array, array + 4, <>(double d : &sum, &factor) ( sum += factor * d ));
for_each(array, array + 4, <&>(double d) ( sum += factor * d ));
```

Note that when the closure stores no copies of variables in the context of the lambda definitions, a different translation is possible. The resulting function object could only store a single pointer to the stack frame where the lambda is defined, and access individual variables via that pointer. We continue, however, to describe the semantics of lambda functions via a translation to function objects with one member variable for each distinct variable that is referred to in the body of the lambda function. Nevertheless, any implementation resulting in the same behavior should be allowed. Section 6.1 discusses the <&> form further, and argues for requiring the "single pointer to environment" closures described above in order to have a fixed size binary representation of lambda functions.

We summarize the rules regarding references to variables defined outside of the lambda function:

- The body of the lambda function can contain references to the parameters of the lambda function, to variables declared in the local variable clause, and to any variable with static storage duration.
- If the lambda function is declared with the <&> form, additionally references to all local variables in scope at the definition of the lambda function are allowed.

# 5 Full translation semantics

We define the semantics of lambda functions via a translation to function objects. Implementations should not, however, be required to literally carry out the translations. We first explain the translation in a case where the body of the lambda function is a single expression and the return type is not specified. Variations are discussed later.

The left-hand side of Figure 1 shows a lambda function in the context of two nested templates. The right-hand side shows the translation. For concreteness of presentation, we fix the example to use two parameters and two local variables. The translation of a lambda function consists of the closure class unique

```
a1 <>(ptype1 param1, ptype2 param2)
a2       : amp1 var1, amp2 var2)
a3      ( body )
```

```
b1  // generated immediately before the statement
b2  // that the lambda expression appears in
b3  class unique {
b4      vtype1-t var1;
b5      vtype2-t var2;
b6      ...
b7  public: // But hidden from user
b8    unique(vtype1-t-param var1, vtype2-t-param var2)
b9      : var1(var1), var2(var2) {}
b10 public: // Accessible to user
b11   unique(const unique& o) : var1(o.var1), var2(o.var2) {}
b12   unique(unique&& o) : var1(move(o.var1)), var2(move(o.var2)) {}
b13
b14   auto operator()(ptype1 param1, ptype2 param2) const
b15     ->decltype( body-t-ret )
b16   { return body-t; }
b17 };
b18
b19 // generated to exactly where the lambda function is defined
b20 unique(init1, init2)
```

Figure 1: Example translation of lambda functions. The ellipses are not part of the syntax but stand for omitted code.

(line b3), generated immediately before the statement where the lambda expression appears, and a call to the constructor of unique to replace the lambda function definition (on line b20). The following list describes the specific points of the translation:

1. The closure object (here named unique) stores the necessary data of the enclosing scope in its member variables. In the example, two variables are stored in the closure: var1 and var2. The type vtype1−t is the type of var1 in the enclosing scope of the lambda expression, augmented with amp1, which is either & or empty. Non-reference non-const members are to be declared mutable—this is to allow updating member variables stored in the closure even though the function call operator of the closure is declared **const**.

2. The closure's constructor has one parameter for each member variable, whose types vtypeN−t−param are obtained from vtypeN−t types by adding **const** and reference, leaving however non-const reference types intact. This constructor should not be exposed to the user.

3. The closure has copy and move constructors with their canonical implementations.

4. Closure classes should not have a default constructor or an assignment operator.

5. The parameter types of the function call operator are those defined in the lambda function's parameter list. The return type is obtained as **decltype**( body−t−ret ) where body−t−ret is obtained from the lambda function's body with a minor translation. The original body may contain references to **this** (implicitly or explicitly), which must be translated to refer to some another unique variable name (see Section 4.2).

6. A lambda function's body may only contain references to its parameters, variables defined in the local variable clause, and any variable with static storage duration. A minor translation for the body is necessary if the lambda function is defined in a member function and contains references to **this**. The translation was described in Section 4.2.

7. The names of all the classes generated in the translation should be unique, different from all other identifiers in the entire program, and not exposed to the user.

The above example translation was a case where the body of the lambda function is a single expression and the return type is not specified. If the return type is specified explicitly, that return type is used as the return type of the function call operator of the closure. The case where the body of the lambda function is a compound statement, instead of a single expression, is only trivially different to the translation described above.

# 6 Binary interface

A closure is of some compiler generated class type. To pass lambda functions to non-template functions, one can use the std::function, or other similar library facilities. For example, the doit function below accepts any function or function object F, including a closure, that satisfies the requirement Callable<F, **int**, **int**>:

```
void doit(std::function<int (int, int)>);
```

The doit function is non-generic, and could be placed, e.g., in a dynamically linked library. The following shows a call to doit, passing a lambda function to it:

```
int i;
 ...
doit(<>(int x, int y : i) ( x + i*y ));
```

The std::function template has a fixed size, and it can be constructed from any MoveConstructible function object which satisfies the callability requirement of the std::function instance. A typical implementation of function uses so called "small buffer optimization" to store small function objects in a buffer in the function itself, and allocate space for larger ones dynamically, storing a pointer in the buffer.

## 6.1 The <&> form

Access to variables declared in the lexical scope of a lambda function can be implemented by storing a single pointer to the stack frame of the function enclosing the lambda. This lends to representing closures using two pointers: one to the code of the lambda function, one to the enclosing stack frame (known as the "static link"). As discussed above, this representation is only safe for lambda functions that do not outlive the functions they are defined in.

If the "two-pointer" representation is mandatory, lambda functions can be given a light-weight binary interface, even more so than using std::function: lambda functions can be passed in registers, and they can be copied bitwise. We put forward the suggestion that lambda functions defined using the <&> syntax are required to use this representation, and that their types are instances of a "magic" template "std::nested_function". To extend the C++ type system with a full-fledged new "lambda-function" type would be rather drastic, which is why we suggest std::nested_function.

The definition of std::nested_function would be similar to that of std::function:

```
template<class MoveConstructible R, class MoveConstructible... ArgTypes>
class function<R(ArgTypes...)>
struct nested_function {
   R operator()(ArgTypes...) const;
   // copy and move constructors
};
```

A function expecting an std::function parameter accepts any function pointer or function object type, assuming the function or function object is callable with the required signature; a function expecting a std::nested_function only matches lambda functions defined using the <&> form. A binary library interface can provide overloads for both types. For example:

```
void doit(std::function<int(int, int)> f);
void doit(std::nested_function<int(int, int)> f);
```

This overload sets allows calls with a function, function object, or lambda function, and takes advantage of the latter overload when called with a `<&>` form lambda function; in a call to `doit` where the parameter is a lambda defined with the `<&>` form, the match to the first requires a user-defined conversion via the templated constructor of `std::function`, whereas the latter is a direct match.

# 7 Acknowledgements

We are grateful for help and comments by Dave Abrahams, Matt Austern, Peter Dimov, Gabriel Dos Reis, Doug Gregor, Andrew Lumsdaine, Valentin Samko, Jeremy Siek, Bjarne Stroustrup, Herb Sutter, Jeremiah Willcock, Jon Wray, and Jeffrey Yasskin.

# References

[JFC07] Jaakko Järvi, John Freeman, and Lawrence Crowl. Lambda functions and closures for C++ (Revision 1). Technical Report N2329=07-0189, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, June 2007.

[JSR06a] Jaakko Järvi, Bjarne Stroustrup, and Gabriel Dos Reis. Decltype (revision 5). Technical Report N1978=06-0048, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, April 2006.

[JSR06b] Jaakko Järvi, Bjarne Stroustrup, and Gabriel Dos Reis. Decltype (revision 6): proposed wording. Technical Report N2115=06-0185, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, November 2006. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2115.pdf`.

[Sam06] Valentin Samko. A proposal to add lambda functions to the C++ standard. Technical Report N1958=06-0028, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, February 2006. `www.open-std.org/JTC1/SC22/WG21/docs/papers/2006/n1958.pdf`.

[Wil07] Anthony Williams. Names, linkage, and templates (rev 1). Technical Report N2187=07-0047, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, March 2007. `www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2187.pdf`.

[WJG+06] Jeremiah Willcock, Jaakko Järvi, Douglas Gregor, Bjarne Stroustrup, and Andrew Lumsdaine. Lambda functions and closures for C++. Technical Report N1968=06-0038, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, February 2006. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1968.pdf`.

# A Proposed wording

The proposed wording follows starting from the next page. Within the proposed wording, text that has been added will be presented in blue and underlined when possible. Text that has been removed will be presented in red, with strike-through when possible. The wording in this document is based on the latest C++0X draft, currently N2284, and uses its LaTeX sources. There are some dangling references in the final document, which will be resolved when merged back to the full sources of the working paper.

# Chapter 3 Lexical conventions [lex]

1 The lexical representation of C++programs includes a number of preprocessing tokens which are used in the syntax of the preprocessor or are converted into tokens for operators and punctuators:

*preprocessing-op-or-punc:* one of

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| { | } | [ | ] | # | ## | ( | ) | |
| <: | :> | <% | %> | %: | %:%: | ; | : | ... |
| new | delete | ? | :: | . | .* | | | |
| + | - | * | / | % | ^ | & | \| | ~ |
| ! | = | < | > | += | -= | *= | /= | %= |
| ^= | &= | \|= | << | >> | >>= | <<= | == | != |
| <= | >= | && | \|\| | ++ | -- | , | ->* | -> |
| <> | <&> | | | | | | | |
| and | and_eq | bitand | bitor | compl | not | not_eq | | |
| or | or_eq | xor | xor_eq | | | | | |

Each *preprocessing-op-or-punc* is converted to a single token in translation phase 7 (**??**).

# Chapter 5 Expressions [expr]

## 5.1 Primary expressions [expr.prim]

Primary expressions are literals, names, ~~and~~ names qualified by the scope resolution operator `::`, and lambda expressions.

*primary-expression:*
    *literal*
    `this`
    `(` *expression* `)`
    *id-expression*
    *lambda-expression*
*id-expression:*
    *unqualified-id*
    *qualified-id*
*unqualified-id:*
    *identifier*
    *operator-function-id*
    *conversion-function-id*
    `~` *class-name*
    *template-id*

### 5.1.1 Lambda Expressions [expr.prim.lambda]

*lambda-expression:*
    *lambda-head exception-specification*$_{opt}$ *lambda-body*

*lambda-head:*
    `<>` `(` *lambda-parameter-declaration-list*$_{opt}$ *lambda-local-var-clause*$_{opt}$ `)`
    `<&>` `(` *lambda-parameter-declaration-list*$_{opt}$ `)`

*lambda-parameter-declaration-list:*
    *lambda-parameter*
    *lambda-parameter* `,` *lambda-parameter-clause*

*lambda-parameter:*
    *decl-specifier-seq declarator*
    *decl-specifier-seq declarator* `=` *assignment-expression*

*lambda-body:*
    *lambda-return-type-clause*$_{opt}$ `(` *expression* `)`
    *lambda-return-type-clause compound-statement*

*lambda-return-type-clause:*
    `->` *type-id*

*lambda-local-var-clause:*
    `;` *lambda-local-var-list*

*lambda-local-var-list:*
    *lambda-local-var*
    *lambda-local-var* `,` *lambda-local-var-list*

*lambda-local-var:*
    `&`$_{opt}$ *identifier*
    `this`

A *lambda-expression* is an *object-expression* and an rvalue. A lambda expression shall have a unique implementation-defined type that is `CopyConstructible`. This type is called the closure type, and the object resulting from evaluating a lambda expression is called a closure object. A closure object can be invoked with the function call syntax with arguments whose types match the *lambda-parameter-declaration-list* of the lambda expression that defines the closure.

1 Closure objects behave as function objects ([function.objects], 20.5), whose defining class's function call operator, constructors, and member variables are defined by the lambda expression's signature, body, and the context of the lambda expression.

In a lambda expression

> `<>(` *lambda-parameter-declaration-list*$_{opt}$ *lambda-local-var-clause*$_{opt}$ `)` *exception-specification*$_{opt}$ `->` *type-id* `(` *compound-statement* `)`

denote the *lambda-parameter-declaration-list*$_{opt}$ as `P`, *lambda-local-var-clause*$_{opt}$ as `L`, *exception-specification*$_{opt}$ as `E`, *type-id* as `R`, and *compound-statement* as `B`.

— The potential scope of a function parameter name in *lambda-parameter-declaration-list* begins at its point of declaration and ends at the end of the lambda expression, but excludes the *lambda-local-var-clause*.

— A name in the *lambda-local-var-clause* shall be in scope in the context of the lambda expression, and must refer to a local object with automatic storage duration or the variable `this`. The same name shall not appear more than once in a *lambda-local-var-clause*.

— A lambda-expression shall not refer to `this` or a variable of automatic storage duration in the enclosing scope of the lambda expression, if it is not named within the lambda expression's *lambda-local-var-list*.

2 The meaning of a lambda function is defined as follows. Define a function object class with a unique name, call it `F`, immediately preceding the statement where the lambda expression occurs.

Define a public function call operator for `F`:

```
auto operator()(P) E' const -> R' B'
```

where `E'`, `R'`, and `B'` are obtained from `E`, `R`, and `B`, respectively, by the following transformations:

— If the lambda expression is within a non-static member function of class `X`, perform name lookup. Transform all accesses to non-static non-type class members of the class `X` or of a base class of `X` that do not use the class member access syntax (5.2.5) to class member access expressions using `(*this)`, as specified in ([class.mfct.non-static], 9.3.1).

— Transform all occurrences of `this` variable to a new variable name, call it `__this`, that does not occur elsewhere in the program.

Let $\mathtt{n}_1, \ldots, \mathtt{n}_k$ denote the names defined in the *lambda-local-var-clause*, where `this` has been translated to `__this`. Let $\mathtt{t}_i, i = 1, \ldots, k$ denote the (non-reference) types of objects denoted by $\mathtt{n}_i$ looked up in the context of the lambda expression; type of `__this` is the type of the `this` variable. If *lambda-local-var-clause* is empty, $k = 0$, and we take the sequences $\mathtt{n}_i$ and $\mathtt{t}_i$ to be empty as well. Define in `F` a private member variable $\mathtt{n}_i$ for each $i = 1, \ldots, k$. If $\mathtt{n}_i$ was declared with `&` in `L`, the type of the member variable $\mathtt{n}_i$ is $\mathtt{t}_i$`&`, otherwise the type is $\mathtt{t}_i$ and the member variable is declared `mutable`.

Define a public copy constructor and a public move constructor for `F` that perform a member-wise copy and move, respectively. Define `F` as a type that is not `Assignable`. Define no other public members to `F`. If the lambda expression is defined in a member function of some class `X`, make `F` a friend of the class `X`.

Replace the lambda expression with an object of type `F` where each member $\mathtt{n}_i$ has been initialized to the value of the variable $\mathtt{n}_i$ and the member `__this` initialized to value of the variable `this`, all looked up in the scope of the lambda expression.

Denote the constructed `F` object `fo`. The meaning of a lambda expression is that of `fo`, except for the following:

— The size and alignment characteristics of the closure object are unspecified. In particular, they are not guaranteed to be those of objects of class `F`.

— If one or more variables in the local variable class are declared with `&`, the effect of invoking a closure object, or its copy, after the innermost function scope of the context of the lambda expression has been exited is undefined.

3 The meaning of a lambda expression of the form

```
<&>( P ) E -> R B
```

where `P` is a *lambda-parameter-declaration-list*$_{opt}$, `E` is an *exception-specification*$_{opt}$, `R` is a *type-id*, and `B` is a *compound-statement*, is that of the lambda expression

```
<>( P L ) E -> R B
```

with `L` the *lambda-local-var-clause* that contains `&var` for each distinct variable name `var` appearing in the body `B` of the lambda expression and denoting a local object with automatic storage duration in the enclosing scope of the lambda expression. If the lambda expression refers to `this`, or contains member accesses that do not mention `this` but would be converted to class member accesses using `(*this)` according to (9.3.1), `L` contains `this`.

The type of a lambda expression of the form `<&>( P ) E -> R B` is `std::nested_function<R(P)>`. [Note: Objects of these types may be implemented with only two words and may provide a stable binary interface.]

4 The meaning of a lambda expression of the form

*lambda-head exception-specification*$_{opt}$ `( E1 )`

where `E1` is an *expression*, is that of the lambda expression

*lambda-head exception-specification*$_{opt}$ `-> decltype(E1) { return E1; }`

5 The meaning of a lambda expression of the form

*lambda-head exception-specification*$_{opt}$ `-> R ( E1 )`

where `E1` is an *expression* and `R` is a *type-id*, is that of the lambda expression

*lambda-head exception-specification*$_{opt}$ `-> R { return E1; }`